CSE 121 Winter 2023 Final Exam

March 14, 2023 - 12:30 – 2:30 PM - KNE 110 & KNE 130

Name of Student: KEY

Section (e.g., AA):_____ Student Number: _____

Do not turn the page until you are instructed to do so.

Rules/Guidelines:

- You must not begin working before time begins, and you must stop working **promptly** when time is called. Any modifications to your exam (writing *or* erasing) before time begins or after time is called will result in a penalty.
- You are allowed one page of notes, no larger than 8.5 x 11 inches. You may not access any other resources or use any electronic devices (including calculators, phones, or smart watches, among others) during the exam. Using unauthorized resources or devices will result in a penalty.
- In general, you are limited to Java concepts or syntax covered in class. You may not use `break`, `continue`, a `return` from a `void` method, `try`/`catch`, or Java 8 features.
- You are limited to the standard Java classes and methods listed on the provided reference sheet. You do not need to write import statements.
- If you abandon one answer and write another, ***clearly cross out*** the answer(s) you do not want graded and ***draw a circle or box*** around the answer you do want graded. When in doubt, we will grade the answer that appears in the space indicated, and the first such answer if there is more than one.
- If you require scratch paper, raise your hand and we will bring some to you.
- If you write an answer on scratch paper, please ***write your name and clearly label*** which question you are answering on the scratch paper, and ***clearly indicate*** on the question page that your answer is on scratch paper. Staple all scratch paper you want graded to the ***end*** of the exam before turning in.
- Answers must be written as proper Java code. Pseudocode or comments will not be graded.
- The exam is not graded on code quality. You are not required to include comments.
- You are also allowed to abbreviate "System.out.print" and "System.out.println" as "S.o.p" and "S.o.pln" respectively. You may **NOT** use any other abbreviations.

Grading:

- Each problem will receive a single E/S/N grade.
    - On problems 1 through 3, earning an E requires answering all parts correctly and earning an S requires answering almost all parts correctly.
    - On problems 4 through 6, earning an E requires an implementation that meets all stated requirements and behaves exactly correctly in *all* cases. Earning an S requires an implementation that meets all stated requirements and behaves exactly correctly in *most* cases or behaves nearly correctly in *all* cases.
- Minor syntax errors will be ignored as long as it is unambiguous what was intended (e.g. forgetting a semicolon, misspelling a variable name where there is only one close option). Major syntax errors, or errors where it is unclear what was intended, may have an impact on your grade.

Advice:

- Read all questions carefully. Be sure you understand the question *before* you begin your answer.
- The questions are not necessarily in order of difficulty. Feel free to skip around. Be sure you are able to at least attempt every question.
- Write clearly and legibly. We cannot award credit for answers we cannot read.
- If you have questions, raise your hand to ask. The worst that can happen is we will say "I can't answer that."
- Ask questions as soon as you have them. Do not wait until you have several questions.

Initial here to indicate you have read and agreed to these rules: _____

1. Code Comprehension

Part A: Check all statements that are true. The entirety of the statement should be true to be selected.

- ☒ Primitive data types in Java (e.g. `int`, `double`, `boolean`, `char`) obey value semantics. This means assigning a new primitive variable to another primitive variable results in the new variable holding a copy of the data type's value.
- ☒ Reference semantics allows us to pass an object as a parameter to a method, and to modify it without need for a `return` statement to "get back" the modified object.
- ☒ Objects in Java (e.g. `Scanner`, `File`, `Random`) obey reference semantics. This means when an object is passed as a parameter to a method, a reference to the object is passed, and no copy of the object is made for the scope of the method.
- ☒ Value semantics implies that any primitive data type variable being passed as a parameter to a method remains unchanged outside the scope of the method call.

Before we begin... difference between reference semantics and value semantics:

Value Semantics

- Primitive Data Types (int, double, boolean, char) are treated by directly manipulating values
- Copy of Data: when you assign one variable to another or pass a variable to a method, a copy of the actual value is made.
- Independence: Changes to 1 variable does not affect the other

→ Ex:
```
int x = 1;   x = 1
int y = x;   x = 1, y = 1   ← y gets a copy of the value of x
y = 2;       x = 1, y = 2   ← changes to y do not affect x
```

→ x isn't changed even though y is

Reference Semantics

- Objects and complex Types: reference semantics are often associated with objects and complex data types
- Reference to Data: Instead of copying the actual data, a reference (memory address) to the data is passed
- Shared data: if you modify the data through one reference, all references pointing to the same data change.

Ex: Value Semantics

```
int[] arr1 = {1,2,3};
int[] arr2 = {1,2,3};   // copies array
arr2[0] = 27;           // changes to arr2 dont impact arr1
S.O.P(Arrays.toString(arr1)); // [1,2,3]
S.O.P(Arrays.toString(arr2)); // [27,2,3]
```

Reference Semantics

```
int[] arr1 = {1,2,3};
int[] arr2 = arr1;      // now, both reference the same array
arr[0] = 27;
S.O.P(Arrays.toString(arr1)); // [27,2,3];
S.O.P(Arrays.toString(arr2)); // [27,2,3];
```

1- Part A

① True- Primitive data types have value semantics. When you assign a new primitive variable to another primitive variable, you are making a copy of the actual value. Changes to one variable do not affect the other.

② True- Reference semantics involves passing a reference to an object. Any modifications made to the object inside the method will affect the original object outside the method. Therefore, there is no need for a return statement to "get back" the modified object.

③ True - Objects in Java, such as instances of classes like Scanners and Randoms, follow reference semantics. When an object is passed as a parameter to a method, it is the reference to the object that is passed, not a copy of the object. Therefore, modifications made to the object inside the method will affect the original object outside the method.

④ True - value semantics for primitive data types mean that when a primitive variable is passed as a parameter to a method, the variables value is passed, and any changes made to the parameter inside the method do not affect the original variable outside the method. Therefore, the primitive data type variable stays unchanged outside the scope of the method call.

Part B: Suppose we want to generate a random integer with an already-declared Random object called `rand`. Which line will give back a random value in the range of 8 (inclusive) to 17 (exclusive)?

- ☐ `int num = rand.nextInt(9) + 17;`
- ☐ `int num = rand.nextInt(8) + 17;`
- ☒ `int num = rand.nextInt(9) + 8;`
- ☐ `int num = rand.nextInt(10) + 8;`

1-Part B
Random syntax can be a bit confusing sometimes. But, the reference sheet shows how to use nextInt if needed.

we want the random to range from 8-16 since 8 is inclusive and 17 is exclusive. Although, by nature,

```
int num = rand.nextInt(exclusive) + inclusive;
```

So, we can add 8 to start the random number range at 8.

```
int num=rand.nextInt(exclusive)+8;
```

Now, since we want numbers from 8-16 and the nextInt value is exclusive, we can do

```
int num=rand.nextInt(9)+8;
```

Since 8+9 = 17 and 17 is exclusive.
NOTE: All of the purple is NOT proper syntax, just for demonstration purposes.

Part C: Trace the evaluation of the following expressions, and give their resulting values. Make sure to give a value of the appropriate type. (i.e. Be sure to include a .0 at the end of a `double` value, or "" around strings.) Write your answer in the box to the right of each expression.

`"c" + (1 + 10 % 4) + "p" + 4 / 9`	"c3p0"
`6 + 5 / 2 * 1.0 - 3 / 2`	7.0
`10 > (18 / 2) && 0 != 10 % 5`	false

1- Part C
We can use our lovely PM(MD)(AS) to help with this problem
1: Parenthesis
2: Mod
3: Multiplication or Division (whichever is first)
4: Addition or Subtraction (whichever is first)

① "c" + (1+10%4) + "p" + 4/9

Parenthesis:
1+10%4 = 1+2
= 3

"c" + 3 + "p" + 4/9

Division:
4/9 = 0 since 4 does not go into 9

"c" + 3 + "p" + 0

Addition:
Since we start with a string, the lone numbers become strings

"c3p0"

② $6 + 5/2 * 1.0 - 3/2$

Division: $5/2 = 2$ since 2 goes into 5 twice

$6 + 2 * 1.0 - 3/2$

Multiplication: $2 * 1.0 = 2.0$ since a double * int = double

$6 + 2.0 - 3/2$

Division: $3/2 = 1$ Since 2 goes into 3 once

$6 + 2.0 - 1$

Addition: $6 + 2.0 = 8.0$ Since double + int = double

$8.0 - 1$

7.0

③ $10 > (18/2) \&\& 0 != 10 \% 5$

Division: $18/2 = 9$

$10 > 9 \&\& 0 != 10 \% 5$

we know the left side is true. Lets check the right side

$true \&\& 0 != 10 \% 5$

Mod: $10 \% 5 = 0$ Since $10/5 = 2$ remainder 0

$true \&\& 0 != 0$

This is false since $0 = 0$

$true \&\& false$

$false$

Anything && with false is false. Anything || with true is true.

2. Array Code Tracing

Consider the following method:

```
public static int[] mystery(int[] list, int ind1, int ind2) {
    int length = (ind2 - ind1) + 1;
    int[] result = new int[length];

    for (int i = 0; i < length; i++) {
        result[i] = list[i + ind1];
    }

    return result;
}
```

Part A: Consider the following code:

```
int[] arr = {4, 9, 3, 6, 8, 3, 4, 0};
int[] result = mystery(arr, 2, 5);
```

What values are present in **`result`** after this code is executed?

- ☐ [4, 9, 4, 0]
- ☒ [3, 6, 8, 3]
- ☐ [4, 9, 2, 6, 8, 5, 4, 0]
- ☐ [4, 9, 6, 8, 4, 0]

2- Part A

we are given

int[] arr = {4,9,3,6,8,3,4,0};

int[] result = mystery(arr, 2, 5);

This tells us our parameters to consider for the method.

public static int[] mystery(int[] list, int ind1, int ind2){

even though this is called list, it represents arr because arr was the actual parameter passed. So, keep in mind:

int[] list = {4,9,3,6,8,3,4,0};

int ind1 = 2;

int ind2 = 5;

```
int ind2 = 5;
int length = (ind2 - ind1) + 1;
int length = (5 - 2) + 1;
           = 3 + 1;
           = 4;
int length = 4;
int[] result = new int[length];
int[] result = new int[4];
→ look something like this
[0, 0, 0, 0]
for(int i = 0; i < length; i++) {   (length → 4)
    result[i] = list[i + ind1];
}
```

To show what is happening at each iteration...

result[0] = list[0 + 2]
= list[2]
= 3

result[1] = list[1 + 2]
= list[3]
= 6

result[2] = list[2 + 2]
= list[4]
= 8

result[3] = list[3 + 2]
= list[5]
= 3

result is now [3, 6, 8, 3]

```
return result;
}
```

Part B: Consider the following code:

```
int[] arr = {4, 9, 3, 6, 8, 3, 4, 0};
int[] result = mystery(arr, 0, arr.length - 1);
```

What values are present in **arr** after this code is executed?

- ☐ [0 9 3 6 8 3 4 7]
- ☒ [4 9 3 6 8 3 4 0]
- ☐ [9 3 6 0 8 3 4]
- ☐ [ ]

2- Part B
Since we discovered in the last problem that mystery returns a new array containing only the elements in the list at indexes between ind1 and ind2 (answer to c) we know the whole list will be returned. This is because the index bounds are between 0 and arr.length-1

Part C: Which of the following best describes what the method `mystery` does?

- ☐ Modify `list` by removing the elements at indexes between `ind1` and `ind2`.
- ☐ Modify `list` by removing all elements except those at indexes between `ind1` and `ind2`.
- ☒ Return a new array containing only the elements in `list` at indexes between `ind1` and `ind2`.
- ☐ Return a new array containing the elements in `list` except those at indexes between `ind1` and `ind2`.

Since int[] list = {4,9,3,6,8,3,4,0}
int ind1=2
int ind2=5
[4,9,3,6,8,3,4,0]
0 1 2 3 4 5 6 7
and the returned result was [3,6,8,3], the third option is correct

while studying...
If you are practicing code tracing and get stumped, plug it into ed and practice de-bugging:

Demo.java

```java
import java.util.*;

public class Demo {
    public static void main(String[] args) {
        int[] arr = {4, 9, 3, 6, 8, 3, 4, 0};
        int[] result = mystery(arr, 2, 5);
        System.out.println(Arrays.toString(result));
    }

    public static int[] mystery(int[] list, int ind1, int ind2) {
        int length = (ind2 - ind1) + 1;
        int[] result = new int[length];

        for (int i = 0; i < length; i++) {
            result[i] = list[i + ind1];
            System.out.println("i: " + i);
            System.out.println("list[i]: " + list[i]);
            System.out.println("list[i + ind1]: " + list[i + ind1]);
        }

        return result;
    }
}
```

/home/Demo.java 15:9 Spaces: 4 (Auto) All changes saved

Console Terminal Run

```
i: 0
list[i]: 4
list[i + ind1]: 3
i: 1
list[i]: 9
list[i + ind1]: 6
i: 2
list[i]: 3
list[i + ind1]: 8
i: 3
list[i]: 6
list[i + ind1]: 3
[3, 6, 8, 3]
```

Program exited with code 0

This can help catch mistakes like writing list[i]+ind1, for example

3. Debugging

Consider a static method called **`findNumber`** that generates random numbers looking for a particular target a particular number of times. The method takes two parameters:

- `int target` - the number to search for
- `int count` - the number of times to find the target

The method generates and prints random integers between 1 and 10 (both inclusive) until the number `target` has been generated `count` times. It then returns how many numbers were generated.

For example, suppose the following call was made:

```
int tries = findNumber(5, 1);
```

This call might produce output like the following:

```
Searching for 1 5s
Generating numbers: 1 8 10 5
```

Step 1: Gather info
- look at parameters and expected output
- Read problem to not miss any edge cases

In the example above, at the end of this call, the value 4 would be returned from the method and stored in the variable `tries`, because 4 numbers were generated before one 5 was seen. (Note that, due to randomness, this call would always not produce the exact same output or return value.)

Consider the following proposed buggy implementation of `findNumber()`:

```
1  public static int findNumber(int target, int count) {
2      Random rand = new Random();
3      int found = 0;
4      int total = 0;
5
6      System.out.println("Searching for " + count + " " + target + "s");
7      System.out.print("Generating numbers: ");
8      while (found < count) {
9          int num = rand.nextInt(10) + 1;
10         System.out.print(num + " ");
11         if (num == count) {
12             found++;
13         }
14         total++;
15     }
16     System.out.println();
17
18     return total;
19 }
```

5 1

found: number of 5s we've found

total: total nums it took to get the correct count(1) of target(5)

while the number of 5s we've found is less than the amount we need

In ex, seeing if 1,8,10, and 5 = 1

count → target

WAIT: Don't we want to see if each of these numbers equal 5?

With the correction, we increase found if we find a 5

return the total

This implementation contains a single bug that is causing it to not work as intended.

(continued on next page…)

(continued on next page…)

Explained on previous page

As an example, if the following code is executed:

```
int tries = findNumber(5, 1);
```

The buggy implementation might produce the following output:

```
Searching for 1 5s
Generating numbers: 2 9 10 8 1
```

After this call to the buggy implementation, the variable `tries` would contain the value 5.

Part A: Identify the single line of code that contains the bug. Write your answer in the box to the right as a single number.

Line 11

Part B: Annotate (write on) the code below to indicate how you would fix the bug. You may add (using arrows to indicate where to insert), remove (by crossing out), or modify (with a combination) any code you choose. However, the fix should not require a lot of work.

```
1  public static int findNumber(int target, int count) {
2      Random rand = new Random();
3      int found = 0;
4      int total = 0;
5
6      System.out.println("Searching for " + count + " " + target + "s");
7      System.out.print("Generating numbers: ");
8      while (found < count) {
9          int num = rand.nextInt(10) + 1;
10         System.out.print(num + " ");
11         if (num == target) {
12             found++;
13         }
14         total++;
15     }
16     System.out.println();
17
18     return total;
19 }
```

4. General Programming

Write a static method named **`matchFirstLetters`** that plays a guessing game with the user. Your method should take two parameters:

- `Scanner console` - the `Scanner` to use for user input
- `String[] passwords` - an array of secret words the user will try to match

Your method should prompt the user to enter a word for each word in `passwords` and inform them if their guess has the same first letter as the password. The user should only be asked for one guess for each word in `passwords`, meaning the total number of guesses will be equal to the number of elements in `passwords`. Your method should then *return* the number of guesses that match.

For example, suppose the following code was executed:

```
Scanner console = new Scanner(System.in);
String[] passwords = {"apple", "banana", "cucumber", "durian", "egg"};
int score = matchFirstLetters(console, passwords);
```

This code would produce output like the following (user input **bold and underlined**):

```
Enter a word: aardvark
It's a match!
Enter a word: duck
No match. The word was banana
Enter a word: cat
It's a match!
Enter a word: zebra
No match. The word was durian
Enter a word: penguin
No match. The word was egg
```

Notice that the user's first guess is tested against the first element of `passwords`, their second guess is tested against the second element, and so on. At the end of this call, the value 2 would be returned from the method and stored in the variable `score`, since the user matched the first letter of 2 words.

You may assume that `passwords` contains at least one value and that the user inputs exactly one word when prompted.

Write your solution in the box on the next page.

Write your solution to problem #4 here:

```
public static int matchFirstLetters(Scanner console, String[] words) {
    int matches = 0;
    for (int i = 0; i < words.length; i++) {
        System.out.print("Enter a word: ");
        String entry = console.next();
        if (entry.charAt(0) == words[i].charAt(0)) {
            matches++;
            System.out.println("It's a match!");
        } else {
            System.out.println("No match. The word was " + words[i]);
        }
    }
    return matches;
}
```

4

Here is how I used to reason through programming problems

① Pay close attention to the parameters and example output (described more in the debugging problem above)

② Determine what is being returned → we need that information when we set up the method.

public static int matchFirstLetters (scanner console, String[] words) {

Because we are instructed to return the number of starting letters, this indicates that we should make a variable to account for this

int matches = 0;
If we re-examine the example output, we see that "Enter a word:" shows up 5 times in a row. This indicates that we might want a loop.
Since we are comparing an entered word to each password in the array...
for(int i = 0; i < words.length; i++){
we print "Enter a word" no matter what the word itself is
S.O.P("Enter a word: ");
you can use S.O.P and S.O.Pln to save time and space.
Now the user enters their guess
String entry = console.next();
We see in the expected output that 1 of 2 things happen each time
→ 1: "It's a match"
→ 2: "No match. The word was" along with the word

Since there are 2 different options, this indicates that we might want an if/else
To establish each condition, we look to see if the first letter of the password and user entered word start with the same letter. Since we want ONE letter, we can use charAt

```
if(entry.charAt(0)== words[i].charAt(0)){
```

words[i].charAt(0) is acceptable syntax because words is an array of strings and words[i] looks at each individual string.
At this point, we know the first letters match, so we increase the number of matches and print

```
matches++;
S.O.pln("It's a match!");
} else {
```

we now know the first letters don't match, so print the alternative

```
S.O.pln("No match. The word was " + words[i]);
}
return matches;
}
```

6. Array Programming

Write a static method called **`sumArrays`** that sums the elements from a pair of arrays. Your method should take two parameters:

- `int[] nums1` - the first array of integers to consider
- `int[] nums2` - the second array of integers to consider

Your method should *return* a new array where each element of the new array contains the sum of the two integers at that index in the parameter arrays. If one array is longer than the other, the result array should contain the extra numbers from the longer array.

For example, suppose the following code was executed:

```
int[] nums1 = {1, 2, 3, 4, 5};
int[] nums2 = {10, 20, 30, 40, 50};
int[] result = sumArrays(nums1, nums2);
```

After this code is executed, the array result would contain the following values:

```
[11, 22, 33, 44, 55]
```

Note that the sum of the integers at each index was put into the result array (for instance 1 plus 10 is 11 and 30 plus 3 is 33).

As another example, suppose the following code was executed:

```
int[] nums1 = {1, 2, 3, 4, 5, 6, 7};
int[] nums2 = {10, 20, 30, 40, 50};
int[] result = sumArrays(nums1, nums2);
```

In this case, after this code is executed, `result` would contain the following values:

```
[11, 22, 33, 44, 55, 6, 7]
```

Notice that the extra elements 6 and 7 from the longer parameter array were also included in the result, but without being summed with anything.

You may assume that both parameter arrays contain at least one element.

Write your solution in the box on the next page.

Write your solution to problem #6 here:

```java
public static int[] sumArrays(int[] nums1, int[] nums2) {
    int[] result = new int[Math.max(nums1.length, nums2.length)];
    for (int i = 0; i < result.length; i++) {
        if (i >= nums1.length) {
            result[i] = nums2[i];
        } else if (i >= nums2.length) {
            result[i] = nums1[i];
        } else {
            result[i] = nums1[i] + nums2[i];
        }
    }
    return result;
}

public static int[] sumArrays2(int[] nums1, int[] nums2) {
    int shorter = Math.min(words1.length, words2.length);
    int longer = Math.max(words1.length, words2.length);
    int[] result = new int[longer];

    for (int i = 0; i < shorter; i++) {
        result[i] = nums1[i] + nums2[i];
    }

    for (int i = shorter; i < longer; i++) {
        if (nums1.length < nums2.length) {
            result[i] = nums2[i];
        } else {
            result[i] = nums1[i];
        }
    }
    return result;
}
```

Second Solution

Before we start, we see that the expected return is [11, 22, 33, 44, 55, 6, 7], or an array of integers. So when we set up our method, we can specify the return type. We are also given int[] nums1={1,2,3,4,5,6,7} and int nums2={10,20,30,40,50} as parameters

```
public static int[] sumArrays(int[] nums1,
int[] nums2){
```

A tricky component to this problem is that the arrays can be different lengths. We also don't know which array is longer. If we tried to jump in immediately and loop over the shorter loop, we wouldn't consider 6 and 7. Or if we looped through the longer array without checking some conditions, we would get an IndexOutOfBounds-exception. We also don't know which is longer or shorter quite yet. So... lets figure this out using math.

```
int shorter= Math.min(nums1.length, nums2.length);
int longer=Math.max(nums1.length,nums2.length);
```

We have now established the length of the shorter array (length of 5) and the longer array (length of 7). If we look at the expected return, we want to return an array that has 7 values.

```
int[] result= new int[longer];
```

Debugging tip: don't forget to specify the size when making a new array.

The goal of the problem is to add values together if they have the same index in each array. Otherwise, add the leftover results to the new array.

① Add values

```
for (int i=0; i<shorter; i++){
```

At this point, we know the variable shorter represents the length of nums2. So, we add these values with the first loop

nums1 = [1, 2, 3, 4, 5, 6, 7]
[10, 20, 30, 40, 50]

and put them in the new array

```
    result[i] = nums1[i] + nums2[i];
}
```

this is what result looks like right now:
[11, 22, 33, 44, 55, 0, 0]
We now need to add 6 and 7 to results

```
for (int i= shorter; i< longer; i++){
```

Even though we know the length of the longer and shorter array at this point, we dont actually know which is which.

```
    if(nums1.length<nums2.length){
        result[i]= nums2[i];
    } else {
        result[i]= nums1[i];
    }
}
```

This establishes which array to get the last 2 values from.

```
return result;
}
```